Efficient Label-Constrained Shortest Path Queries on Road Networks: A Tree Decomposition Approach

Junhua Zhang
Nanjing University of Science and Technology
University of Technology Sydney
junhua.zhang@student.uts.edu.au

Long Yuan*
Nanjing University of Science and Technology
longyuan@njust.edu.cn

Wentao Li
University of Technology Sydney
wentao.li@uts.edu.au

Lu Qin
University of Technology Sydney
lu.qin@uts.edu.au

Ying Zhang
University of Technology Sydney
ying.zhang@uts.edu.au

ABSTRACT

Computing the shortest path between two vertices is a fundamental problem in road networks. Most of the existing works assume that the edges in the road networks have no labels, but in many real applications, the edges have labels and label constraints may be placed on the edges appearing on a valid shortest path. Hence, we study the label-constrained shortest path queries in this paper. In order to process such queries efficiently, we adopt an index-based approach and propose a novel index structure, LSD-Index, based on *tree decomposition*. With LSD-Index, we design an efficient query processing algorithm with good performance guarantees. Moreover, we also propose an algorithm to construct LSD-Index and further improve the efficiency of index construction by exploiting the parallel computing techniques. We conduct extensive performance studies using large real road networks including the whole USA road network. Compared with the state-of-the-art approach, the experimental results demonstrate that our algorithm not only achieves up to 2 orders of magnitude speedup in query processing time but also consumes much less index space. Meanwhile, the indexing time is also competitive, especially that for the parallel index construction algorithm.

PVLDB Reference Format:
Junhua Zhang, Long Yuan, Wentao Li, Lu Qin, Ying Zhang. Efficient Label-Constrained Shortest Path Queries on Road Networks: A Tree Decomposition Approach. PVLDB, 15(3): 686-698, 2022.
doi:10.14778/3494124.3494148

1 INTRODUCTION

Computing the shortest path between two locations is one of the fundamental problems in road networks [3, 8, 9, 12, 14, 20, 22, 28, 30, 34, 40]. In real road networks, not all roads are the same, for example, highways allow faster travel, toll roads cost money, and the transport of hazardous goods is forbidden on roads in water protection areas. Therefore, many applications place constraints on the edges appearing on a valid shortest path when computing the shortest path, which leads to the study of label-constrained shortest path queries [10, 25]. Formally, given a road network G where each edge has a label, a source vertex s, a target vertex t, and an edge label set $\mathcal{L}$, a label-constrained shortest path query $q = (s, t, \mathcal{L})$ aims to compute the shortest path from s to t such that the labels of edges on the shortest path are contained in $\mathcal{L}$.

Label-constrained shortest path queries can be used in many real application scenarios such as personal routine planing [25] and emergency evacuation navigation [16]. For example, the shortest path from Irvine, CA to Riverside, CA travels along State Route 261, which is a local toll road through this area. For a user who does not wish to pay the toll fee, we can find the shortest path from Irvine to Riverside that actually avoids all toll roads by a label-constrained shortest path query $q = (\text{"}Irvine\text{"}, \text{"}Riverside\text{"}, \mathcal{L})$ in which $\mathcal{L}$ does not contains the label representing toll road [25]. In China, new drivers who get the driver license in less than 12 months are not allowed to drive cars on expressway alone for safety [31]. Therefore, the expressway should be avoided when planning routines for these new drivers, which can be achieved by the label-constrained shortest path queries where $\mathcal{L}$ does not contain the label representing expressway. In emergency evacuation navigation, the recommended evacuation route should avoid the roads in dangerous areas [16], which can be achieved by the label-constrained shortest path queries where $\mathcal{L}$ does not contain the label representing roads in dangerous areas.

Motivation. A straightforward approach for label-constrained shortest path queries is to use Dijkstra's algorithm [7] by only visiting the edges whose edge label is in $\mathcal{L}$ during the traversal. Although this approach can compute the required shortest path, as the road network is large in real applications, it cannot satisfy the real-time requirements for the label-constrained shortest path queries as it may traverse the whole road network when s and t are far away from each other. As a result, researchers resort to index-based techniques to accelerate the label-constrained shortest path query processing [10, 25].

The state-of-the-art index-based approach is edge-disjoint partition (EDP) [10]. Intuitively, EDP partitions the road network based on each edge label and caches the computed shortest path information for processed queries in each partition as the index structure.

* Long Yuan is the corresponding author.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.

Proceedings of the VLDB Endowment, Vol. 15, No. 3 ISSN 2150-8097.
doi:10.14778/3494124.3494148

When a new query comes, the cached information is used to accelerate the query processing. Obviously, the performance of EDP heavily depends on the hit ratio of the index. However, there are no theoretical guarantees on the hit ratio of EDP, as it just caches the computed shortest path in each partition for the processed queries, but the newly issued queries may distribute diversely and the label-constrained shortest path for a specific query maybe involve several partitions. Consequently, it is quite possible that the hit ratio for a specific query is low and EDP degenerates into an online search algorithm similar to direct using Dijikstra's algorithm. Even worse, the performance of EDP could be poorer than direct using Dijikstra's algorithm as more vertices may be visited due to the introduction of the index.

Motivated by this, in this paper, we re-investigate the label-constrained shortest path queries on road networks and aim to design an efficient label-constrained shortest path query processing algorithm with non-trivial theoretical performance guarantees.

Our approach. We also resort to index-based techniques to accomplish efficient label-constrained shortest path query processing. As tree decomposition can decompose a road network into a tree-like structure with small treeheight and treewidth, it achieves great success in computing the shortest path on unlabelled road networks recently [15, 21]. Inspired by this, we revisit the tree decomposition based indexing techniques for shortest path problem.

We start from the shortest path queries on *unlabelled road networks*. Regarding this problem, the state-of-the-art tree decomposition based indexing approach processes a query with time complexity $O(h \cdot \omega^2)$, where h, ω is the treeheight, treewidth of the tree decomposition, respectively [30]. By carefully analyzing the properties of the tree decomposition, we present an algorithm based on the tree decomposition for the shortest path queries, and non-trivially prove that the time complexity of the algorithm to process a query can be bounded by $O(h \cdot \omega)$, which reduces the time complexity of [30] by a factor ω (refer to Theorem 4.13). Since h and ω are small for road networks, it means we can efficiently process the shortest path queries on unlabelled road networks based on tree decomposition with theoretical performance guarantees.

Based on the above findings, we explore the tree decomposition based indexing solution for label-constrained shortest path queries. A direct indexing solution is as follows: for each induced road network by one possible combination of the edge label set in Σ, where Σ is the finite alphabet used for the labels of edges in G, we treat the induced road network as an unlabelled road network and build the tree decomposition based index for it. Given a label-constrained shortest path $q = (s, t, \mathcal{L})$, we retrieve the corresponding index for the edge label set $\mathcal{L}$ and compute the shortest path accordingly. Clearly, this approach fully utilizes the efficiency of the tree decomposition based indexing technique for shortest path queries on unlabelled road networks. However, the total number of indices constructed in this approach is $2^{|\Sigma|}$. It is prohibitive to construct and maintain such a number of indices, which makes this approach unscalable to handle large road networks in real applications.

Observing the indices constructed in the direct solution, we find that lots of redundant information regarding label-constrained shortest path computation are stored among different indices. Following this observation, we conceive of reducing the redundant information among these $2^{|\Sigma|}$ indices and integrating them into a holistic compact index structure. To make our idea practically applicable, the following issues need to be addressed: (1) how to design such an index that the redundant information is reduced while the efficiency of query processing is not compromised? (2) how to efficiently construct the index for a road network, especially when the road network is large?

Table 1: List of Notations

Notation	Description
$G = (V, E)$	graph G with vertex set V and edge set E
$\phi(\cdot)$, $\ell(\cdot)$	weight and label of an edge/path
Σ	alphabet of edge labels
$G[\Sigma_s]$	Σ_s-induced subgraph of G
$\mathrm{dist}_G^{\mathcal{L}}(s, t)$	label-constrained shortest distance
T_G	tree decomposition of G
$X, X(v)$	tree node
$\omega(T_G)/\omega, h(T_G)/h$	treewidth and treeheight of T_G
$\mathcal{S}, \mathcal{S}(u, v)$	label-constrained shortest distance set(LSDS)
ρ	the maximum size of LSDS
$\mathcal{G}_i$	label-constrained distance preserved graph

Contribution. In this paper, we address the above issues and make the following contributions:

(1) A new tighter bound on the shortest path query processing on unlabelled road networks. We revisit the problem of tree decomposition based indexing for shortest path queries on unlabelled road networks and present an algorithm for this problem. We prove the time complexity of the algorithm is $O(h \cdot \omega)$, while the state-of-the-art tree decomposition based indexing approach is $O(h \cdot \omega^2)$.

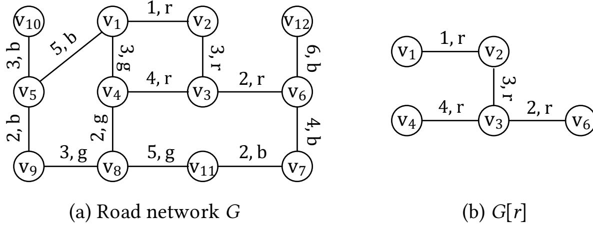
(2) Efficient algorithms for label-constrained shortest path queries with theoretical performance guarantees. We design a new tree decomposition based index for the label-constrained shortest path queries. Based on the index, we propose an algorithm to answer the queries. We also design an algorithm to construct the index. Moreover, considering the road networks in real applications could be very large, we exploit parallel computing techniques to further speed up the construction of the index.

(3) Extensive performance studies on real road networks. We conduct extensive performance studies on eight real large road networks including the whole road network of the USA. The experimental results demonstrate the efficiency and effectiveness of our index.

Proofs and part of experimental results are omitted in this paper due to limited space, they can be found in our technical report [39].

2 PRELIMINARIES

Let $G = (V, E, \phi, \ell, \Sigma)$ be a labelled road network, where $V(G)$ is a set of vertices, $E(G)$ is a set of edges, $\phi : E(G) \rightarrow \mathbb{R}^+$ is a function that assigns each edge $e \in E(G)$ a positive number $\phi(e, G)$ as its weight, Σ is a finite alphabet of edge labels, and $\ell : E(G) \rightarrow \Sigma$ is a function assigns each edge $e \in E(G)$ a label $\ell(e, G) \in \Sigma$. We use $n = |V(G)|$ (resp. $m = |E(G)|$) to denote the number of vertices (resp. edges) in G. For each vertex $v \in V(G)$, the neighbors of v, denoted by $\mathrm{nbr}(v, G)$, is defined as $\mathrm{nbr}(v, G) = \{u | (u, v) \in E(G)\}$. The degree of a vertex v is the number of neighbors of v. Given



(a) Road network G (b) $G[r]$

Figure 1: A Road Network and Label-induced Subgraph

a subset of labels $\Sigma_s \subseteq \Sigma$, the Σ_s-induced subgraph of G, denoted by $G[\Sigma_s]$, is the subgraph contains all edges in G with labels in Σ_s. A path p in G is a sequence of vertices $p = (v_0, v_1, v_2 ... v_k)$, where $(v_i, v_{i+1}) \in E(G)$ for each $0 \leq i \leq k-1$. We use $P(s, t, G)$ to denote all paths from s to t. The weight of the path, denoted by $\phi(p, G)$, is defined as $\phi(p, G) = \sum_{0 \leq i \leq k-1} \phi(v_i, v_{i+1})$. Given two vertices s and t, the shortest path from s to t is the path with minimum weight in $P(s, t, G)$. The shortest distance, denoted by $\text{dist}_G(s, t)$, is the weight of the shortest path between s and t. For a given path p in G, the label of p, denoted by $\ell(p, G)$, is the union of edge labels in p, i.e., $\ell(p, G) = \bigcup_{e \in p} \ell(e, G)$. For simplicity, we omit G in the notations if the context is self-evident. For ease of reference, we summarize the frequently used notations in Table 1.

Definition 2.1. **(Label-Constrained Path)** Given two vertices s, t in a road network $G = (V, E, \phi, \ell, \Sigma)$ and a set of edge labels $\mathcal{L} \subseteq \Sigma$, a path from s to t is a label-constrained path regarding $\mathcal{L}$ if $\ell(p) \subseteq \mathcal{L}$.

Definition 2.2. **(Label-Constrained Shortest Path)** Given two vertices s, t in a road network $G = (V, E, \phi, \ell, \Sigma)$ and a set of edge labels $\mathcal{L} \subseteq \Sigma$, the label-constrained shortest path from s to t is the path with the minimum weight among the label-constrained paths from s to t regarding $\mathcal{L}$.

Problem statement. Given a road network $G = (V, E, \phi, \ell, \Sigma)$, a label-constrained shortest path query is defined as $q = (s, t, \mathcal{L})$, where $s, t \in V(G)$, $\mathcal{L} \subseteq \Sigma$, and the answer is the label-constrained shortest path from s to t regarding $\mathcal{L}$. In this paper, we aim to develop effective indexing techniques to answer q efficiently.

For the ease of explanation, we first consider that G is undirected, and discuss how to handle directed road networks in Section 5.5.

Example 2.3. Consider the road network G in Figure 1 (a), the weight and the label of each edge is shown beside the corresponding edges. For example, $\phi((v_1, v_2)) = 1$ and $\ell((v_1, v_2)) = r$. For an edge label set $\{r\}$, the $\{r\}$-induced subgraph $G[r]$ is shown in Figure 1 (b), which consists of edges with label r. Given vertices v_5, v_6 and a label set $\{b, g\}$, there are two label-constrained paths between v_5 and v_6: $\{(v_5, v_1, v_4, v_8, v_{11}, v_7, v_6), (v_5, v_9, v_8, v_{11}, v_7, v_6)\}$ and the second one is the label-constrained shortest path with weight 16.

3 EXISTING SOLUTION

Edge-disjoint partitioning (EDP) [10] is the state-of-the-art solution for the label-constrained shortest path queries. EDP is an index-based approach consisting of two components:

EDP **indexing.** Given a road network G, EDP first partitions G by the labels of edges. For each label $l \in \Sigma$, the partition Part_l contains the edges with label l, i.e., $\text{Part}_l = G[l]$. It is clear that each edge label uniquely corresponds to a partition, we use them interchangeably when the context is self-evident. Based on the partitions, a vertex v in a partition Part_{l_i} is a *bridge vertex* if there exists an edge $(v, u) \in \text{Part}_{l_j}$, and $l_i \neq l_j$. For a bridge vertex $v \in \text{Part}_{l_i}$, its *OtherHosts* is other partitions containing v. When processing queries, it computes the shortest paths in each partition. These computed paths are all cached in the EDP index. As more queries are processed, which leads to the index size exceeds a specified threshold, EDP uses the least recently used (LRU) replacement strategy to replace the old paths with new computed shortest paths.

Query processing. For a query $q = (s, t, \mathcal{L})$, EDP adopts a greedy traversal paradigm similar to Dijkstra's algorithm to compute the label-constrained shortest path. During the traversal, it maintains a min-priority queue Q, each element of Q has three attributes: (1) Part: the identifier of a partition, (2) v: a vertex id, and (3) d: currently observed distance from s to v. Q is keyed by (Part, v) and ordered by d. EDP initially inserts $(\text{Part}_{l_s}, s, 0)$ into Q, where Part_{l_s} is the partition in which s resides. Then, EDP iteratively extracts elements e' from Q, expands the traversal, and inserts frontier discovered vertex into Q until t is reached or Q becomes empty. During the expansion, EDP first computes the shortest distances d from $e'.v$ to bridge vertices v' in e'.Part, then, for each Part $\in \{\mathcal{L}' \cap \mathcal{L}\}$, where $\mathcal{L}'$ represents the labels of $v'.OtherHosts$ in e'.Part, it inserts (Part, v', $d + e'.d$) to Q (if t is in e'.Part, the same procedure is applied to t as well). When computing the distances from $e'.v$ to a bridge vertices v' in a partition, EDP directly obtains it if it is already cached in the index; Otherwise, it performs Dijkstra's algorithm and caches the result in the index.

Drawbacks of EDP**.** EDP processes the label-constrained shortest path queries correctly, but it has the following two drawbacks in efficiency: (1) theoretically, there is no non-trivial tight bound on its query processing time. The worst-case time complexity of EDP is not better than the online search following Dijkstra's algorithm. (2) practically, EDP just caches the computed shortest paths in each partition for the processed queries, but the newly issued queries may distribute diversely and the label-constrained shortest path for the query may involve several partitions, it is quite possible that most of the needed information for a specific query is not cached. In this case, EDP degenerates into an online traversal based algorithm similar to the Dijikstra's algorithm.

4 A NAIVE INDEXING APPROACH

4.1 Tree Decomposition

Tree decomposition [26] decomposes a graph into a tree-like structure to speed up solving graph problems, it is defined as:

Definition 4.1. **(Tree Decomposition)** Given a graph G, a tree decomposition T_G of G is a rooted tree with nodes $\{X_1, \cdots, X_n\}$, where each node is a subset of $V(G)$ (i.e., $X_i \subseteq V(G)$), such that:

(1) $\bigcup_{X \in V(T_G)} X = V(G)$;
(2) for each edge $(u, v) \in E(G)$, there is a node $X \in T_G$ such that $u \in X$ and $v \in X$;
(3) for each $v \in V(G)$, the nodes containing v (i.e., $\{X | v \in X\}$) form a connected subtree of T_G.

Definition 4.2. **(Treewidth and Treeheight)** Given a tree decomposition T_G of G, the treewidth of T_G, denoted by $\omega(T_G)$ is

one less than the maximum size of all nodes in T_G, i.e., $\omega(T_G) = \max_{X \in V(T_G)} |X| - 1$. The treeheight of T_G, denoted by $h(T_G)$, is the maximum depth (the depth of a node in T_G is the distance from the node to the root node of T_G) of all nodes in T_G.

For ease of presentation, we refer to $v \in V(G)$ in G as a vertex and refer to $X \in V(T_G)$ in T_G as a node. We use ω and h to denote the treewidth and treeheight of the tree decomposition T_G if the context is self-evident. The treewidth of a graph G is the minimum treewidth over all possible tree decompositions of G.

To determine whether a given graph G has treewidth at most a given variable is NP-Complete [2]. Existing techniques to compute the optimal tree decomposition can only handle small graphs [13]. Thus, we adopt a suboptimal but practically effective algorithm, MDE, to conduct the tree decomposition [35].

Minimum degree elimination based tree decomposition. MDE conducts the tree decomposition in two steps: (1) it iteratively eliminates a vertex v with the minimum degree in G, and then adds edges between all neighbors of v, v's neighbors form a clique in G. Clearly, after the elimination of v, v's neighbors become its neighbor's neighbor. It proceeds the elimination until G becomes empty. For each elimination, the eliminated vertex v and its neighbors $\mathsf{nbr}(v)$ form a node $X(v)$ in T_G. (2) After all the vertices are eliminated, for each node $X(v)$, $X(u)$ is set as the parent of $X(v)$ in T_G, where $X(u)$ is the node created by the first eliminated vertex u in $X(v) \setminus \{v\}$.

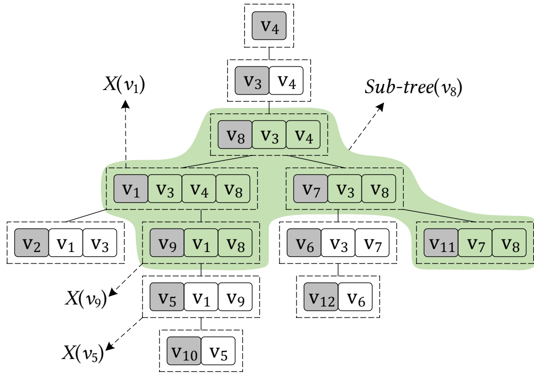


Figure 2: A Tree Decomposition T_G of G

Example 4.3. Figure 2 shows the tree decomposition T_G of G in Figure 1 (a) generated by MDE. T_G has 12 nodes. The vertex elimination order is v_{10}, v_{12}, v_5, v_2, v_6, v_{11}, v_7, v_9, v_1, v_8, v_3, v_4. The elimination of a vertex v leads to a unique node $X(v)$ in T_G. For example, the elimination of v_5 creates node $X(v_5) = \{v_5, v_1, v_9\}$. The nodes that contains v_8 form a connected subtree of T_G (the green area). Since the nodes in T_G contain at most 4 vertices, the treewidth $\omega = 3$, and treeheight $h = 6$.

4.2 A Naive Indexing Approach

Given $G = (V, E, \phi, \ell, \Sigma)$, there are $2^{|\Sigma|}$ possible edge label combinations. Therefore, we can build $2^{|\Sigma|}$ indices and each index is built upon the induced subgraph by one possible combination of the edge label in Σ. As all the possible edge label combinations are considered, for each index, we only need to treat the corresponding induced subgraph as unlabelled and build the index following the shortest path indexing technique for the unlabelled road networks. To answer a query $q = (s, t, \mathcal{L})$, the index for $\mathcal{L}$ is utilized to retrieve the shortest path. Following this idea, we present a naive indexing approach based on tree decomposition.

Before presenting the naive indexing approach, we first introduce the vertex cut property of the tree decomposition, this property is the key to apply tree decomposition to shortest path queries.

Definition 4.4. **(Vertex Cut)** Given a graph G, a subset of vertices $C \subset V(G)$ is a vertex cut of G if the deletion of C from G splits G into multiple connected components. Given two vertices s and t in G, the vertex cut C is a s-t cut if the deletion of C from G disconnects s and t, and we say C separates s and t.

Lemma 4.5. *[27] Given a tree decomposition T_G of G, for any non-root node X_c and its parent X_p, if exists $s \in X_c \setminus X_p$ and $t \in X_p \setminus X_c$, then $X_c \cap X_p$ is a vertex cut of G and separates s and t.*

Lemma 4.6. *[5] Given a tree decomposition T_G of G, for any two vertices s and t in $V(G)$, suppose $X(s)$ is not an ancestor/decedent of $X(t)$ in T_G, let X_{lca} be the lowest common ancestor (LCA) of $X(s)$ and $X(t)$ in T_G, then X_{lca} is a vertex cut of G and separates s and t.*

Given a s-t cut C, it is obvious that every path from s to t passes at least one vertex in C. Accordingly, we have:

Lemma 4.7. *[21] Given two vertices s and t in G, let C be a s-t cut, then* $\mathsf{dist}(s, t) = \min_{v \in C}\{\mathsf{dist}(s, v) + \mathsf{dist}(v, t)\}$.

Example 4.8. Consider the tree decomposition T_G of G shown in Figure 2. For $X(v_9)$ and its parent node $X(v_1)$, $X(v_9) \cap X(v_1) = \{v_1, v_8\}$ is a vertex cut of G, which separates v_9 and v_3. As shown in Figure 2, for $X(v_{10})$ and $X(v_{12})$, their LCA is $X(v_8)$, we know $\mathsf{dist}(v_{10}, v_3) = 12, \mathsf{dist}(v_{10}, v_4) = 10, \mathsf{dist}(v_{10}, v_8) = 8; \mathsf{dist}(v_{12}, v_3) = 8$, $\mathsf{dist}(v_{12}, v_4) = 12$ and $\mathsf{dist}(v_{12}, v_8) = 14$, the shortest distance from v_{10} to v_{12} is $\mathsf{dist}(v_{10}, v_{12}) = \min(12 + 8, 10 + 12, 8 + 14) = 20$.

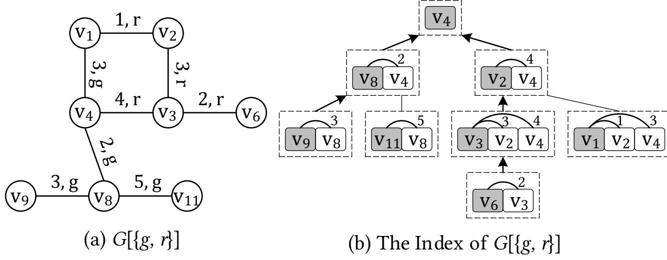
Since the label-constrained shortest path between two vertices can be easily obtained if their label-constrained shortest distance is determined with our algorithms, we focus on the *computation of the label-constrained shortest distance* between two vertices hereafter for clearness and discuss how to obtain the corresponding shortest path in Section 5.4. For brevity, given two vertices u, v, and an edge label set $\mathcal{L} \subseteq \Sigma$, we use $\mathsf{dist}_G^{\mathcal{L}}(u, v)$ to denote the label-constrained shortest distance from u to v regarding $\mathcal{L}$ in G.

The naive indexing approach. Based on the above lemmas, we can devise a straightforward indexing approach to compute label-constrained shortest distance between two vertices as follows:

- *Indexing.* For each possible edge label set $\Sigma_s \subseteq \Sigma$, we first retrieve the Σ_s-induced subgraph $G[\Sigma_s]$. Based on $G[\Sigma_s]$, we compute the tree decomposition $T_{G[\Sigma_s]}$ with MDE. After that, for each $X(v) \in T_{G[\Sigma_s]}$, we compute the $\mathsf{dist}_{G[\Sigma_s]}(v, u)$ for any $u \in X(v) \setminus \{v\}$ and store them in node $X(v)$ using hash table. Note that we also maintain the mapping from vertex v to node $X(v)$ in the index for the ease of query processing.
- *Query Processing.* Given a query $q = (s, t, \mathcal{L})$, we can compute $\mathsf{dist}_G^{\mathcal{L}}(s, t)$ based on the index $T_{G[\mathcal{L}]}$ built on $G[\mathcal{L}]$. The detailed procedure is shown in Algorithm 1. It first computes the lowest common ancestor X_{lca} of $X(s)$ and $X(t)$ in $T_{G[\mathcal{L}]}$ (line 1). After

Algorithm 1: NaiveQuery $(s, t, \mathcal{L}, T)$

1 $X_{\mathsf{lca}} \leftarrow$ find LCA of $X(s)$ and $X(t)$ in $T_{G[\mathcal{L}]}$;
`// compute shortest distance from s to vertices in` X_{lca}
2 $d_s(\cdot) \leftarrow \infty$, $d_s(s) \leftarrow 0$;
3 **foreach** $w \in X(s) \backslash \{s\}$ **do**
4 $\quad d_s(w) \leftarrow \mathsf{dist}_{G[\mathcal{L}]}(s, w)$;
5 $X'_s \leftarrow X(s)$;
6 **while** $X_{\mathsf{lca}} \neq X'_s$ **do**
7 $\quad X_p \leftarrow$ parent of X'_s in $T_{G[\mathcal{L}]}$;
8 $\quad$ **for** $u \in X_p \setminus X'_s$ **do**
9 $\quad\quad$ **for** $v \in X_p \cap X'_s$ **do**
10 $\quad\quad\quad d_s(u) = \min\{d_s(u), d_s(v) + \mathsf{dist}_{G[\mathcal{L}]}(v, u)\}$;
11 $\quad X'_s \leftarrow X_p$;
12 Repeat line 2-11 by replacing s with t;
13 **return** $\min_{w \in X_{\mathsf{lca}}} \{d_s(w) + d_t(w)\}$;



(a) $G[\{g, r\}]$ (b) The Index of $G[\{g, r\}]$

Figure 3: The Naive Indexing Approach

that, it computes the distance from s to vertices in X_{lca}. Based on Lemma 4.5, for a node X'_s and its parent X_p, where X'_s is an ancestor node of $X(s)$, and assume that the shortest distances from s to all vertices in X'_s are already computed, then the shortest distances from s to vertices in $u \in X_p \setminus X'_s$ can be calculated as $\mathsf{dist}_{G[\mathcal{L}]}(s, u) = \min_{w \in X'_s \cap X_p}\{\mathsf{dist}_{G[\mathcal{L}]}(s, w) + \mathsf{dist}_{G[\mathcal{L}]}(w, u)\}$, where $\mathsf{dist}_{G[\mathcal{L}]}(w, u)\}$ can be accessed by looking up the hash table in $X(w)$ or $X(u)$. Hence, we can iteratively compute the shortest distances from s to vertices in X_{lca} along the tree path from X_s to X_{lca} (line 3-11). The distances from t to vertices in X_{lca} can be computed similarly (line 12). Finally, $\mathsf{dist}^{\mathcal{L}}_G(s, t)$ is obtained via the vertices in X_{lca} based on Lemma 4.6 and Lemma 4.7 (line 13).

Example 4.9. Reconsider the road network shown in Figure 1 (a). Figure 3 (a) shows the $\{g, r\}$-induced subgraph $G[\{g, r\}]$. Figure 3 (b) shows the corresponding index $T_{G[\{g,r\}]}$ built on $G[\{g, r\}]$. For the node in $T_{G[\{g,r\}]}$, such as $X(v_1) = \{v_1, v_2, v_4\}$, we store $\mathsf{dist}_{G[\{g,r\}]}(v_1, v_2) = 1$ and $\mathsf{dist}_{G[\{g,r\}]}(v_1, v_4) = 3$ in it. For a query $q = (v_9, v_6, \{g, r\})$, the arrows in Figure 3 (b) demonstrate the query processing procedure. The LCA of $X(v_9)$ and $X(v_6)$ is $X(v_4)$. It iteratively computes the shortest distance from v_9 and v_6 to the vertices in $X(v_4)$ following the arrows. For example, when computing the shortest distances from v_9 to $v_4 \in X(v_8)$, as $X(v_9) \cap X(v_8) = v_8$, $\mathsf{dist}_{G[\{g,r\}]}(v_9, v_4) = \mathsf{dist}_{G[\{g,r\}]}(v_9, v_8) + \mathsf{dist}_{G[\{g,r\}]}(v_8, v_4) = 5$. Similarly, it computes $\mathsf{dist}_{G[\{g,r\}]}(v_6, v_4) = 6$. Hence, $\mathsf{dist}_{G[\{g,r\}]}(v_9, v_6) = \mathsf{dist}_{G[\{g,r\}]}(v_9, v_4) + \mathsf{dist}_{G[\{g,r\}]}(v_6, v_4) = 11$.

Theorem 4.10. *Given a query $q = (s, t, \mathcal{L})$, Algorithm 1 computes $\mathsf{dist}^{\mathcal{L}}_G(s, t)$ correctly.*

Lemma 4.11. *Given a tree decomposition T_G of G generated by* MDE, *for a node X of T_G, $|X \bigcup_{X_a \in \mathcal{A}(X)} X_a| \leq h$, where $\mathcal{A}(X)$ represents the ancestors of X in T_G.*

Lemma 4.12. *Given a tree decomposition T_G of G, for a node $X(v)$ and a non-root ancestor node $X(u)$ of $X(v)$, let $X_p(v)$ (resp. $X_p(u)$) be the parent node of $X(v)$ (resp. $X(u)$), then $\{X_p(v) \setminus X(v)\} \cap \{X_p(u) \setminus X(u)\} = \emptyset$.*

Theorem 4.13. *Given a query $q = (s, t, \mathcal{L})$, Algorithm 1 computes $\mathsf{dist}^{\mathcal{L}}_G(s, t)$ in $O(h \cdot \omega)$.*

Remark. TEDI [30], the state-of-the-art tree decomposition based indexing approach for the shortest path queries on unlabelled road networks, presents a query processing algorithm using a similar idea as Algorithm 1 with time complexity $O(h \cdot \omega^2)$. As shown in Theorem 4.13, our presented algorithm has a time complexity of $O(h \cdot \omega)$, which reduces that of TEDI by a factor ω.

5 OUR NEW INDEXING APPROACH

As shown in our experiments (Table 2), MDE generally generates a tree decomposition with small treeheight h and treewidth ω for road networks. For example, h and ω for the whole USA road network is 2,886 and 579, respectively. Hence, Algorithm 1 permits an efficient query processing regarding a label-constrained query. However, the naive approach needs to construct $2^{|\Sigma|}$ separate indices. Obviously, it is prohibitive to construct and maintain such $2^{|\Sigma|}$ separate indices. In this section, we exploit the dominance relationships between edge-labelled paths and present a new index based on the tree decomposition. The new index can overcome the problem of the naive index with little additional cost for the query processing.

5.1 A New Index Structure

Reconsider the road network G in Figure 1, due to the existence of path $\{v_1, v_2, v_3, v_6\}$, the shortest distance between v_1 and v_6 in $\{b, g, r\}$-induced subgraph is 5. Meanwhile, in $\{b, r\}$, $\{g, r\}$ and $\{r\}$ induced subgraphs, the shortest distance between v_1 to v_6 is 5 as well. In this case, if we have already stored the shortest distance 5 between v_1 and v_6 in the index constructed on $G[\{r\}]$, it is redundant to store the same information in the indices constructed on $G[\{b, r\}]$, $G[\{g, r\}]$, and $G[\{b, g, r\}]$. Based on this observation, instead of considering all the possible edge label sets regarding Σ separately, we can treat these possible edge label sets as a whole and design a holistic compact index that covers all the shortest distance information without storing any redundant information. Following this idea, we have the following lemma:

Lemma 5.1. *Given two vertices u, v in G, let p be a path between u and v in G, then u can reach v in distance d regarding a label set $\mathcal{L}$ if $\ell(p) \subseteq \mathcal{L}$, $\phi(p) \leq d$.*

Following Lemma 5.1, we define the label-constrained shortest distance set between two vertices as follows:

Definition 5.2. **(Label-constrained Shortest Distance Set)** Given a road network G and two vertices u and v in G, the label-constrained shortest distance set (LSDS) of u, v, denoted by $\mathcal{S}(u, v)$, is a set of label-distance pairs $\{(L_1, d_1), (L_2, d_2), \dots\}$ such that:

(1) For each $(L_i, d_i) \in \mathcal{S}(u, v)$, there exists a path p from u to v with $L_i = \ell(p)$ and $d_i = \phi(p)$.

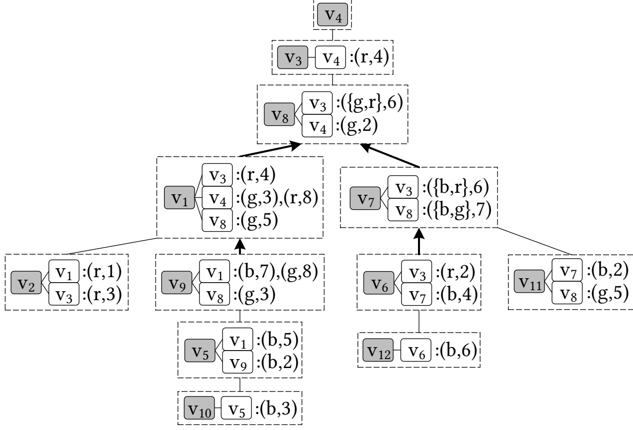


Figure 4: The LSD-Index

(2) For any path p from u to v, there exists a $(L_i, d_i) \in \mathcal{S}(u,v)$, $L_i \subseteq \ell(p)$ and $d_i \leq \phi(p)$;
(3) For any path p from u to v and $(L_i, d_i) \in \mathcal{S}(u,v)$, if $\ell(p) \subset L_i$, then $d_i < \phi(p)$; if $\ell(p) = L_i$, then $d_i \leq \phi(p)$.

Condition (1) ensures that each label-distance pair corresponds to a path in G. Condition (2) guarantees that $\mathcal{S}(u,v)$ covers all possible label-constrained shortest distances between u and v. Condition (3) ensures that the set is minimum and there is no redundancy label-distance pair regarding label-constrained shortest distance according to Lemma 5.1. Based on Definition 5.2, for a given G and its tree decomposition T_G, we construct the label-constrained shortest distance index as follows:

Definition 5.3. **(Label-constrained Shortest Distance Index)** Given a road network G, let T_G be its tree decomposition, the label-constrained shortest distance index of G, denoted by LSD-Index, is built on T_G. For each node $X(v) \in V(T_G)$, the label-constrained shortest distance sets from v to other vertices $u \in X(v) \setminus \{v\}$ are precomputed and stored.

Example 5.4. Figure 4 shows the LSD-Index of G in Figure 1 (a). Each node stores the corresponding LSDS. Take $\mathcal{S}(v_1, v_4)$ as an example. $\mathcal{S}(v_1, v_4) = \{(g,3),(r,8)\}$ because (1) $(g,3)$ and $(r,8)$ correspond to path (v_1, v_4) and path (v_1, v_2, v_3, v_4) between v_1 and v_4, respectively, (2) any other paths between v_1 and v_4, such as $(v_1, v_5, v_9, v_8, v_4)$, can be covered by these two paths, and (3) there is no redundancy in $\{(g,3),(r,8)\}$.

5.2 Query Processing by LSD-Index

With LSD-Index, we can easily obtain a query processing algorithm similar to Algorithm 1. The details are shown in Algorithm 2. Given a query $q = (s, t, \mathcal{L})$, Algorithm 2 first computes the lowest common ancestor X_{lca} of $X(s)$ and $X(t)$ (line 1). Then, it computes the label-constrained distance from s (resp. t) to the vertices in X_{lca} along the tree path similarly to Algorithm 1 (line 3-11). Finally, it computes the label-constrained shortest distance by iterating over vertices in X_{lca} (line 13). Procedure dist computes the label-constrained shortest distance between u and v regarding an edge label set L, it iterates the label-distance pair (L', d') in $\mathcal{S}(u,v)$ (line 15) and returns the shortest distance d' such that $L' \subseteq L$ (line 16-17).

Algorithm 2: LSD-Index-Query $(s, t, \mathcal{L}$, LSD-Index $T)$

1 $X_{\text{lca}} \leftarrow$ find LCA of $X(s)$ and $X(t)$ in T;
   // compute LSD from s to vertices in X_{lca}
2 $d_s^{\mathcal{L}}(\cdot) \leftarrow \infty$, $d_s^{\mathcal{L}}(s) \leftarrow 0$;
3 **foreach** $w \in X(s) \setminus \{s\}$ **do**
4     $d_s^{\mathcal{L}}(w) \leftarrow \texttt{dist}(s, w, \mathcal{L})$;
5 $X'_s \leftarrow X(s)$;
6 **while** $X_{\text{lca}} \neq X'_s$ **do**
7     $X_p \leftarrow$ parent of X'_s in T;
8   **for** $u \in X_p \setminus X'_s$ **do**
9   **for** $v \in X_p \cap X'_s$ **do**
10            $d_s^{\mathcal{L}}(u) \leftarrow \min(d_s^{\mathcal{L}}(u), d_s(v) + \texttt{dist}(v, u, \mathcal{L}))$;
11    $X'_s \leftarrow X_p$;
12 Repeat line 2-11 by replacing s with t;
13 **return** $\min_{w \in X_{\text{lca}}}(d_s^{\mathcal{L}}(w) + d_t^{\mathcal{L}}(w))$;

14 **Procedure** $\texttt{dist}(u, v, L)$
    // assume $\mathcal{S}(u,v)$ is ordered by distance
15   **for** $(L', d') \in \mathcal{S}(u,v)$ **do**
16   **if** $L' \subseteq L$ **then**
17   **return** d';

Example 5.5. Reconsider the query $q = (v_9, v_6, \{g, r\})$, the arrows in Figure 4 demonstrate the query process. The LCA of $X(v_9)$ and $X(v_6)$ is $X(v_8)$. For v_9, $\text{dist}_G^{\{g,r\}}(v_9, v_1) = 8$, as $\mathcal{S}(v_9, v_1)$ contains $(g, 8)$. Similarly, $\text{dist}_G^{\{g,r\}}(v_9, v_8) = 3$. Following the arrow, $X(v_9) \cap X(v_1) = \{v_1, v_8\}$ and $X(v_1) \setminus X(v_9) = \{v_3, v_4\}$. Hence $\text{dist}_G^{\{g,r\}}(v_9, v_3) = \min\{\text{dist}_G^{\{g,r\}}(v_9, v_1) + \text{dist}_G^{\{g,r\}}(v_1, v_3), \text{dist}_G^{\{g,r\}}(v_9, v_8) + \text{dist}_G^{\{g,r\}}(v_8, v_3)\} = 9$. Similarly, $\text{dist}_G^{\{g,r\}}(v_9, v_4) = 5$. For v_6, $\text{dist}_G^{\{g,r\}}(v_6, v_3) = 2$, $\text{dist}_G^{\{g,r\}}(v_6, v_4) = 6$, and $\text{dist}_G^{\{g,r\}}(v_6, v_8) = 8$ can be computed similarly. Then, $\text{dist}_G^{\{g,r\}}(v_9, v_6) = \min_{w \in \{v_3, v_4, v_8\}} \{\text{dist}_G^{\{g,r\}}(v_9, w) + \text{dist}_G^{\{g,r\}}(w, v_6)\} = 11$.

Theorem 5.6. *Given a road network G, the size of the* LSD-Index *is $O(n \cdot \omega \cdot \rho)$, where ρ represents the maximum size of* LSDS *stored in* LSD-Index.

Theorem 5.7. *Given a query $q = (s, t, \mathcal{L})$, Algorithm 2 computes $\text{dist}_G^{L}(s,t)$ in $O(h \cdot \omega \cdot \rho)$.*

Remark. Compared with the naive approach, an additional factor ρ is introduced in the time complexity of Algorithm 2. However, as shown in our experiments (Table 2), ρ is very small in practice. On the other hand, due to LSD-Index, our approach avoids constructing and maintaining $2^{|\Sigma|}$ separate indices in the naive approach.

5.3 LSD-Index Construction

To construct the LSD-Index, a direct solution is based on Definition 5.3 as follows: we first conduct the tree decomposition on G, and then compute the LSDS for the vertices in each node according to Definition 5.2. In this approach, the time complexity for the LSDS computation is $O(n \cdot \omega \cdot ((2^{|\Sigma|})^2 + 2^{|\Sigma|} \cdot (m + n \log n)))$. Obviously, the cost of this part is prohibitive, which consequently makes this approach impractical.

To address this problem, we propose a new index construction algorithm. Instead of dividing the construction into two independent procedures, the new algorithm progressively maintains partial

Algorithm 3: LSDS Operators

```
1  Procedure LSDSJoin(S'_p, S''_p)
2  |  S_p ← ∅;
3  |  foreach (L', d') ∈ S'_p do
4  |  |  foreach (L'', d'') ∈ S''_p do
5  |  |  |  S_p ← S_p ∪ {(L' ∪ L'', d' + d'')};
6  |  return S_p;
7  Procedure LSDSPrune(S_p)
8  |  foreach (L, d) ∈ S_p do
9  |  |  foreach (L', d') ∈ S_p do
10 |  |  |  if L ⊆ L' and d ≤ d' then
11 |  |  |  |  S_p ← S_p \ {(L'_c, d'_c)};
12 |  return S_p;
```

Algorithm 4: LSD-Index-Cons(G)

```
   // phase 1
1  G_0 ← G; T ← ∅ ;
2  foreach (u, v) ∈ G do
3  |  S_p(u, v) ← {(ℓ((u, v)), φ((u, v)))};
4  for i ← 1 to n do
5  |  v ← vertex in G_{i−1} with minimum degree;
6  |  π(v) ← i; G_i ← G_{i−1} \ v;
7  |  foreach u, w ∈ nbr(v, G_{i−1}) do
8  |  |  S' ← LSDSJoin(S_p(v, u), S_p(v, w));
9  |  |  if (u, w) ∉ G_{i−1} then
10 |  |  |  add an edge (u, w) into G_i;
11 |  |  |  S_p(u, w) ← LSDSPrune(S');
12 |  |  else
13 |  |  |  S_p(u, w) ← LSDSPrune(S_p(u, w) ∪ S');
14 |  X(v) ← {v} ∪ nbr(v, G_{i−1});
15 foreach X(v) ∈ T do
16 |  if π(v) < n then
17 |  |  u ← the vertex in X(v)\{v} with smallest π(·) ;
18 |  |  add X(v) as the child of X(u);
   // phase 2
19 for i ← n − 1 to 1 do
20 |  v ← vertex with π(·) = i ;
21 |  for u ∈ X(v) \ {v} do
22 |  |  for w ∈ X(v) \ {v, u} do
23 |  |  |  S' ← LSDSJoin(S_p(v, w), S_p(u, w));
24 |  |  |  S_p(v, u) ← LSDSPrune(S_p(v, u) ∪ S');
```

LSDS by coordinating the procedures of the tree decomposition and LSDS computation. Based on the partial LSDS, the new algorithm computes the complete LSDS in a top-down manner in which the computed complete LSDS can be re-used to accelerate the computation for those not-yet-computed complete LSDS. Before presenting the algorithm, we first introduce two operators on LSDS that are used in the index construction algorithm:

Definition 5.8. **(Operator** LSDSJoin**)** Given two LSDS $\mathcal{S}'_p$ and $\mathcal{S}''_p$, operator LSDSJoin generates a new LSDS by joining the entities in $\mathcal{S}'_p$ and $\mathcal{S}''_p$, i.e., $\text{LSDSJoin}(\mathcal{S}'_p, \mathcal{S}''_p) = \{(L' \cup L'', d' + d'') \mid \forall (L', d') \in \mathcal{S}'_p \wedge (L'', d'') \in \mathcal{S}''_p\}$

Definition 5.9. **(Operator** LSDSPrune**)** Given a LSDS $\mathcal{S}_p$, operator LSDSPrune removes (L_i, d_i) from $\mathcal{S}_p$, if $\exists\, (L_j, d_j) \in \mathcal{S}_p$ such that $L_j \subseteq L_i \wedge d_j \leq d_i$, where $i \neq j$.

Algorithm. With the above operators, our new index construction algorithm is shown in Algorithm 4. It contains two phases: in phase 1, it conducts tree decomposition in which partial LSDS are computed for vertex pairs incident to the involved edges (line 1-18); in phase 2, it computes the complete LSDS in a top-down manner based on the partial LSDS of phase 1 (line 19-24).

• *Phase 1: Partial* LSDS *maintained tree decomposition.* In phase 1, it conducts the tree decomposition following MDE and maintains the partial LSDS for the vertex pairs incident to edges involved in the decomposition. Specifically, it first initializes G_0 as G and T as an empty tree (line 1). For each edge (u, v), $\mathcal{S}_p(u, v)$ is initialized as $\{(\ell((u, v)), \phi((u, v)))\}$, where $\mathcal{S}_p(u, v)$ is used to store the partial LSDS (line 2-3). After that, it performs vertex elimination iteratively following the procedure of MDE (line 4-14). In the ith iteration, it eliminates the vertex v with minimum degree from G_{i-1} and assigns its $\pi(\cdot)$ as i, where $\pi(\cdot)$ records the elimination order (line 5-6). Then, for each vertex pair u, w in the $\text{nbr}(v, G_{i-1})$, it first joins $\mathcal{S}_p(v, u)$ and $\mathcal{S}_p(v, w)$ by LSDSJoin and obtains $\mathcal{S}'$ (line 8). If G_{i-1} does not contain an edge (u, w), it adds an edge (u, w) into G_i and assigns $\mathcal{S}_p(v, w)$ as the result of LSDSPrune on $\mathcal{S}'$ (line 9-11); Otherwise, the $\mathcal{S}_p(u, w)$ is updated as the result of LSDSPrune on $\mathcal{S}_p(u, w) \cup \mathcal{S}'$ (line 13). In Algorithm 4, we assume $\pi(u) < \pi(w)$ for the clearness of the presentation. After the elimination of v, it adds a node $X(v)$ containing v and its neighbors $\text{nbr}(v, G_{i-1})$ into T (line 14). After all vertices are eliminated, the parent-child relationships between nodes are generated (line 15-18). For a non-root vertex v, it selects the vertex $u \in X(v) \setminus \{v\}$ with smallest $\pi(\cdot)$ value (line 17) and sets $X(u)$ as the parent node of $X(v)$ (line 18).

Before presenting phase 2, we first introduce the label-constrained shortest distance (LSD) preserved graph $\mathcal{G}_i$ whose properties form the theoretical foundation of phase 2 and are also used for the proof of Algorithm 4.

Definition 5.10. **(**LSD **Preserved Graph)** Given a graph G_i generated in phase 1, the LSD preserved graph of G_i, denoted by $\mathcal{G}_i$, is a labelled multigraph such that (1) $V(\mathcal{G}_i) = V(G_i)$; (2) if there is an edge $e = (u, v)$ in G_i, then, for each entity $(L, d) \in \mathcal{S}_p(u, v)$, there is an edge $e' = (u, v)$ with $\ell(e') = L$ and $\phi(e') = d$ in $\mathcal{G}_i$.

Lemma 5.11. *Given two vertices u, v in $\mathcal{G}_i$, for any edge label set L, $\text{dist}^L_{\mathcal{G}_i}(u, v) = \text{dist}^L_G(u, v)$.*

According to Lemma 5.11, the label-constrained shortest distance between any two vertices in G_i is preserved in $\mathcal{G}_i$. Moreover, we have the following lemma:

Lemma 5.12. *Given a vertex $v \in V(\mathcal{G}_{\pi(v)-1})$, for any edge label set L, the label-constrained shortest path from v to $u \in X(v) \setminus \{v\}$ regarding L in $\mathcal{G}_{\pi(v)-1}$ only contains v and u, or passes a vertex in $X(v) \setminus \{v, u\}$.*

Therefore, for a vertex v, if we have already known the complete LSDS $\mathcal{S}(u, w)$ for any two vertices $u, w \in X(v) \setminus \{v\}$, to compute the complete LSDS $\mathcal{S}(v, u)$, according to Definition 5.2, Lemma 5.11 and Lemma 5.12, we only need to join the partial LSDS $\mathcal{S}_p(v, w)$ with the complete $\mathcal{S}(u, w)$ for each $w \in X(v) \setminus \{v, u\}$ with LSDSJoin, add the result to $\mathcal{S}_p(v, u)$ and remove redundant label-distance pair with LSDSPrune. Moreover, we can apply the above procedure

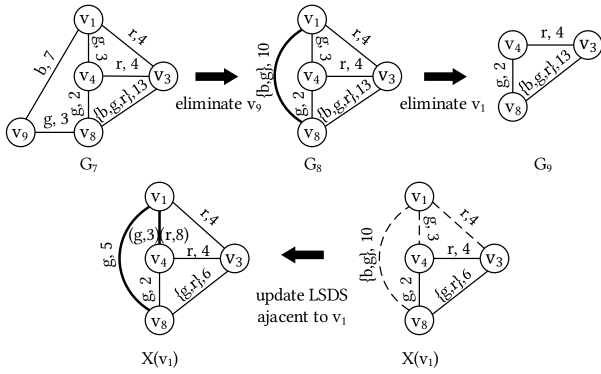


Figure 5: Procedure of Index Construction

recursively to compute the complete LSDS $\mathcal{S}(u, w)$. Following this idea, we can compute the complete LSDS in a top-down manner based on tree decomposition and use the computed complete LSDS to accelerate the computation of not-yet-computed complete LSDS.

• *Phase 2: Top-down complete* LSDS *computation.* The phase 2 of the construction algorithm is shown in line 19-24 of Algorithm 4. It processes the vertices in the decreasing order of their $\pi(\cdot)$ value (line 19). For each vertex v, to compute the complete LSDS of v and u, where $u \in X(v) \setminus \{v\}$, it iterates the vertices $w \in X(v) \setminus \{v, u\}$, computes $\mathcal{S}'$ by LSDSJoin on $\mathcal{S}_p(v, w)$ and $\mathcal{S}_p(u, w)$, and removes the redundancy in $\mathcal{S}_p(v, u) \cup \mathcal{S}')$ with LSDSPrune (line 22-24). The construction finishes when all the vertices are processed.

Example 5.13. In Figure 5, the upper (resp. lower) part illustrate some of key steps of phase 1 (resp. phase 2) during the construction of LSD-Index for G in Figure 1 (a), where the LSDS is shown near each edge. For example, in phase 1, when eliminating v_9 from G_7, a new edge (v_1, v_8) is added, and $\mathcal{S}_p(v_1, v_8) = \{(\{b, g\}, 10)\}$ is obtained by joining $\mathcal{S}_p(v_9, v_8)$ and $\mathcal{S}_p(v_9, v_1)$). In phase 2, for $\mathcal{S}(v_1, v_8)$, since LSDSJoin$(\mathcal{S}_p(v_1, v_4), \mathcal{S}(v_8, v_4)) = \{(\{g\}, 5), (\{g, r\}, 10)\}$, $(\{g, r\}, 10)$ and $(\{b, g\}, 10) \in \mathcal{S}_p(v_1, v_8)$ are redundant because of $(\{g\}, 5)$, thus, $\mathcal{S}_p(v_1, v_8)$ is updated to $\{(\{g\}, 5)\}$. Similarly, $\mathcal{S}(v_1, v_4)$ is updated to $\{(\{g\}, 3), (\{r\}, 8)\}$.

Theorem 5.14. *Given a road network G, the time complexity of Algorithm 4 to construct the index is $O(n \cdot \omega^2 \cdot \rho^2)$.*

5.4 Shortest Path Restoration

The algorithms described in the previous section focus on computing the label-constrained shortest distance. By slightly modifying the index structure and query processing algorithm, we can easily retrieve the corresponding label-constrained shortest path.

Augmented LSD-Index**.** According to Definition 5.2, each entity $(L, d) \in \mathcal{S}(u, v)$ in LSD-Index corresponds to a path $p(u, v)$ in G. To restore the shortest path for a query, we first need to restore the path represented by (L, d). Revisiting the construction procedures of LSD-Index shown in Algorithm 4, there are two cases in which $(L, d) \in \mathcal{S}(u, v)$ is generated: (1) the original edge (u, v) in G; (2) operator LSDSJoin is applied on $\mathcal{S}_p(w, u)$ and $\mathcal{S}_p(w, v)$ (line 8 or line 23). For case (1), we do not store any additional information in $\mathcal{S}(u, v)$. For case (2), we store (w, id_u, id_v) besides (L, d) in $\mathcal{S}(u, v)$, where id_u (resp. id_v) is the identification of (L_u, d_u) (resp. (L_v, d_v)) in $\mathcal{S}(w, u)$ (resp. $\mathcal{S}(w, v)$) that leads to the generation of (L, d). With this additional information, we can restore the path $p(u, v)$ represented by (L, d) in $\mathcal{S}(u, v)$ as follows: (1) $p(u, v)$ is the original edge (u, v) in G; or (2) $p(u, v)$ can be obtained by concatenating $p(u, w)$ and $p(w, v)$ represented by (L_u, d_u) in $\mathcal{S}(w, u)$ and (L_v, d_v) in $\mathcal{S}(w, v)$, respectively, while $p(u, w)$ and $p(w, v)$ can be obtained recursively in the same way. Clearly, as the size of added information for each entity is constant, the space complexity of the augmented LSD-Index and the time complexity of the corresponding construction algorithm keep the same as that for LSD-Index.

Query processing. For query processing, the general framework is similar to Algorithm 2 but with additional path information. Specifically, we keep the shortest paths p (resp. p') from s (resp. t) to the vertices in X_{lca} by storing the vertex v and the corresponding $(L, d) \in \mathcal{S}(v, u)$ leading to the final $d_s^{\mathcal{L}}(u)$ in line 4 and line 10 of Algorithm 2, and concatenate p and p' through $w \in X_{\text{lca}}$ which leads to the final shortest distance in line 11 of Algorithm 2. For the edges in p (resp. p') that are not the original edges of G (represents by $(L, d) \in \mathcal{S}(v, u)$), they can be restored by the method as discussed above. Given a $q = (s, t, \mathcal{L})$, if the returned shortest path p has τ edges, then, the extra time complexity to restore the path can be bounded by $O(\tau)$. Since the lower bound to answer q is $\Omega(\tau)$ and τ is generally very small compared with $h \cdot \omega \cdot \rho$, the time complexity of the query processing is the same as that of Algorithm 2.

5.5 Extension for Directed Road Networks

In previous sections, we assume the road networks are undirected . Our techniques can be extended to support directed road networks.

Indexing. For the index structure, the LSD-Index for directed road networks is similar to that for undirected road network with two differences: (1) for the tree decomposition, we extend MDE for the directed road networks as follows: it iteratively eliminates the vertex v with the minimum degree and connects any pair u, w of v's neighbors with directed edges after the elimination of v, and the other parts are the same. (2) for LSDS stored in each node T_G, we trivially extend the label-constrained distances defined in Definition 5.2 for directed road networks by using the paths with direction. And for each node $X(v)$, we pre-compute and store $\mathcal{S}{<}v, u{>}$ and $\mathcal{S}{<}u, v{>}$ for any $u \in X(v) \setminus \{v\}$. Here, we use $\mathcal{S}{<}v, u{>}$ to represent the LSDS from v to u extended for directed road networks for distinction. For the index construction algorithm, the whole framework is the same as Algorithm 4 except the directions of edges/paths need to be considered.

Query processing. The query processing procedure for directed road networks is similar to Algorithm 2. Given a query $q = (s, t, \mathcal{L})$, we first compute the lowest common ancestor X_{lca} of $X(s)$ and $X(t)$. After that, we compute the label-constrained shortest distances from s to vertices in X_{lca} and from these vertices to t. Finally, we can obtain the label-constrained shortest distance from s to t and consequently restore the label-constrained shortest path from s to t in the same way as discussed for the undirected road networks.

5.6 Handling Large Σ

Although our indexing techniques can significantly reduce the index size, Σ might be very large in some scenarios, which makes the index size still very large. In this section, we introduce how to extend our techniques to address this issue.

It has been widely observed that the labels in real-life graphs usually follows the power-law distribution [24]. Therefore, we treat the high frequent labels and low frequent labels in different ways. Let Σ_f be the set of high frequent labels in G. We create a set of virtual labels Σ_v by evenly partitioning the labels in $\Sigma \setminus \Sigma_f$ into $|\Sigma_v|$ groups and each virtual label represents the labels in each group, where $|\Sigma_f| + |\Sigma_v| \ll |\Sigma|$. In G, we replace the real low frequent label for each edge with the corresponding virtual label and construct the LSD-Index regarding $\Sigma_f \cup \Sigma_v$.

Given a query $q = (s, t, \mathcal{L})$, if $\mathcal{L} \subseteq \Sigma_f$, we use Algorithm 2 to answer the query directly. Otherwise, let $\mathcal{L}_f$ be the label set $\mathcal{L} \cap \Sigma_f$ and $\mathcal{L}_v$ be the virtual label set representing labels in $\mathcal{L} \cap \{\Sigma \setminus \Sigma_f\}$. We compute $\text{dist}_G^{\mathcal{L}_f}(s, t)$ and $\text{dist}_G^{\mathcal{L}_f \cup \mathcal{L}_v}(s, t)$ following Algorithm 2 based on the index. Clearly, $\text{dist}_G^{\mathcal{L}_f}(s, t) \geq \text{dist}_G^{\mathcal{L}}(s, t)$ and $\text{dist}_G^{\mathcal{L}}(s, t) \geq \text{dist}_G^{\mathcal{L}_f \cup \mathcal{L}_v}(s, t)$. Thus, if $\text{dist}_G^{\mathcal{L}_f}(s, t) = \text{dist}_G^{\mathcal{L}_f \cup \mathcal{L}_v}(s, t)$, we obtain $\text{dist}_G^{\mathcal{L}}(s, t)$. Otherwise, the shortest path may involve some edges with virtual labels in $\mathcal{L}_v$ but real labels not in $\mathcal{L}$. In this case, for index entries $(L_v, d_v) \in \mathcal{S}(u, v)$ that are used for obtaining $\text{dist}_G^{\mathcal{L}_f \cup \mathcal{L}_v}(s, t)$ and contain virtual labels, we need to further check whether $d_v = \text{dist}_G^{\mathcal{L}}(u, v)$, this can be achieved by exploring the neighbors w of u connected with labels in $\mathcal{L}$ and recursively computing $\text{dist}_G^{\mathcal{L}}(w, v)$. If $d_v \neq \text{dist}_G^{\mathcal{L}}(u, v)$, we use the refined $\text{dist}_G^{\mathcal{L}}(u, v)$ instead and the correct final result can be obtained.

6 PARALLEL INDEX CONSTRUCTION

Although Algorithm 4 significantly reduces the time cost to construct LSD-Index compared with building the index directly based on the definition, it is still expensive for large road networks due to the inevitable LSDSJoin and LSDSPrune operations during the LSDS computation. In this section, we further improve the construction efficiency by parallelizing the LSDS computation.

Recall that the computation of LSDS contains the partial LSDS maintenance in phase 1 and top-down complete LSDS computation in phase 2. For the partial LSDS maintenance in phase 1, the computation of $\mathcal{S}_p(v, u)$ in $X(v)$ only depends on $\mathcal{S}_p(w, v)$ and $\mathcal{S}_p(w, u)$ in $X(w)$, where $X(w)$ is a descendant of $X(v)$ in the tree decomposition. For the top-down complete LSDS computation in phase 2, the computation of $\mathcal{S}_p(v, u)$ in $X(v)$ only depends on $\mathcal{S}_p(v, w)$ and $\mathcal{S}_p(w, u)$ in $X(w)$, where $X(w)$ is an ancestor of $X(v)$ in the tree decomposition. Hence, we define:

Definition 6.1. **(Tree Decomposition Level)** Given a tree decomposition T of G, for a node $X(v)$, the tree decomposition level of $X(v)$, denoted by $l(X(v))$, is defined as $l(X(v)) =$

$$\begin{cases} \min\{l(X(u)) | X(u) \in X(v).\text{children}\} + 1, & X(v).\text{children} \neq \emptyset \\ 1, & X(v).\text{children} = \emptyset \end{cases}$$

where $X(v)$.children represents the children of $X(v)$ in T.

As discussed above, if we compute the LSDS level by level based on Definition 6.1 (from bottom level to top level in phase 1 while from top level to bottom level in phase 2), then the LSDS computations related to the nodes at the same level has no dependence with each other, which means we can process the computations related to these nodes simultaneously with any extra costs.

Algorithm 5: LSD-Index-ParCons(G)

```
1  G_0 ← G; T ← ∅;
2  for i ← 1 to n do
3      line 5-6 of Algorithm 4;
4      foreach u, w ∈ nbr(v, G_{i-1}) do
5          insert v into D(u, w);
6          line 9-10 of Algorithm 4;
7      X(v) ← {v} ∪ nbr(v, G_{i-1});
8  line 15-18 of Algorithm 4;
9  for i ← 1 to n do
10     v ←vertex with π(·) = i;
11     if X(v).children = ∅ then l(X(v)) ← 1;
12     else l(X(v)) ← min_{X(u)∈X(v).children} l(X(u)) + 1 ;
13 l_max ← max_{X(v)∈T} l(X(v));
   // Partial LSDS computation
14 for i ← 1 to l_max do
15     for X(v) ∈ T with l(X(v)) = i in parallel do
16         for u ∈ X(v) in parallel do
17             if (v, u) ∈ G then
18                 S_p(v, u) ← {(ℓ((v, u)), ϕ((v, u)))};
19             else S_p(v, u) ← ∅ ;
20             for w ∈ D(v, u) do
21                 S' ← LSDSJoin(S_p(w, v), S_p(w, u));
22                 S_p(v, u) ← LSDSPrune(S_p(v, u) ∪ S');
   // Complete LSDS computation
23 for i ← l_max to 1 do
24     for X(v) ∈ T with l(X(v)) = i in parallel do
25         for u ∈ X(v) \ {v} in parallel do
26             for w ∈ X(v) \ {v, u} do
27                 S' ← LSDSJoin(S_p(v, w), S_p(w, u));
28                 S_p(v, u) ← LSDSPrune(S' ∪ S_p(v, u));
```

Algorithm. Following the above idea, the parallel construction algorithm, LSD-Index-ParCons, is shown in Algorithm 5. LSD-Index-ParCons follows a similar framework to Algorithm 4. It first conducts the tree decomposition following MDE (line 1-8). During the decomposition, instead of maintaining the partial LSDS, it only records the vertex v leading to the update of $\mathcal{S}_p(u, w)$ in $\mathcal{D}(u, w)$ (line 5). After finishing the tree decomposition, it computes the tree decomposition level for each nodes following Definition 6.1 (line 9-13). Then, it conducts partial LSDS computation in a bottom-up manner (line 14-22) and the complete LSDS computation in a top-down manner (line 23-28). For the nodes at a specific level, they are processed simultaneously (line 15-16, line 24-25). When the algorithm finishes, LSD-Index is correctly constructed, which can be proved similar to Algorithm 4.

7 EXPERIMENTS

In this section, we compare our algorithms with the state-of-the-art methods for label-constrained shortest path queries. All experiments are conducted on a machine with an Intel Xeon 2.5GHz CPU (40 cores) and 256 GB main memory running Linux.

Table 2: Datasets used in Experiments

Dataset	Description	n	m	$\|\Sigma\|$	h	ω	ρ	ρ_{avg}	Indexing Time (S)	Indexing Time (P)	Index Size
NY	New York City	264,346	733,846	10	717	126	28	1.29	36.74s	6.97s	34.52 MB
COL	Colorado	435,666	1,057,066	10	477	133	32	1.12	24.27s	4.81s	36.45 MB
FLA	Florida	1,070,376	2,712,798	10	643	82	38	1.19	34.03s	5.76s	95.91 MB
CAL	California	1,890,815	4,657,742	10	834	177	31	1.11	92.04s	15.93s	161.21 MB
EST	Eastern USA	3,598,623	8,778,114	10	1,366	240	28	1.17	258.63s	39.79s	327.17 MB
WST	Western USA	6,262,104	15,248,146	10	1,450	299	35	1.11	343.51s	45.18s	546.95 MB
CTR	Central USA	14,081,816	34,292,496	10	2,342	540	94	1.31	5,959.00s	741.12s	1.45 GB
USA	Full USA	23,947,347	58,333,344	10	2,886	570	136	1.27	7,152.18s	903.94s	2.37 GB

Datasets. We use eight publicly available real road networks from DIMACS [1]. In each road network, vertices represent intersections between roads, edges correspond to roads or road segments, the weight of an edge is the physical distance between two vertices, and the label of an edge represents its road types. The road types of these road networks can be divided into four main categories: (1) A1, Primary Highway With Limited Access; (2) A2, Primary Road Without Limited Access; (3) A3, Secondary and Connecting Road; (4) A4, Local, Neighborhood, and Rural Road. The road types follows the power-law distribution. Since different datasets contain different number of labels (from $18 \sim 32$), in our experiments (except Exp-6), for the purpose of controlling variables and keeping the distribution of labels as same as possible, we refine the labels of each dataset and make each dataset contain 10 labels using the following method: for the labels in each main category, we sort the labels in the increasing order of their frequency and merge two labels with similar frequency as one, we continue this process until only 10 labels remains. Table 2 provides the details about these datasets. Table 2 shows the value of h and ω of the tree decomposition for each road network and it is clear that h and ω are small in practice. Table 2 also shows the value of ρ and ρ_{avg} of LSD-Index for each road network, where ρ_{avg} represents the average size of LCDS in LSD-Index. It is clear that ρ and ρ_{avg} are much smaller than h and ω in practice.

Algorithms. We compare the following algorithms. All the algorithms are implemented in C++ and compiled in GCC 8.3.1 with -O3 flag. We adopt OpenMP to implement our parallel algorithm.

- Dijkstra: direct online search algorithm using the Dijkstra's algorithm following the edges with labels in given $\mathcal{L}$.
- EDP: The state-of-the-art algorithm for label-constrained shortest path queries, which is introduced in Section 3.
- LSD-Index: Our proposed algorithms include query processing algorithm (Algorithm 2), index construction algorithm (Algorithm 4), and parallel index construction algorithm (Algorithm 5).

For EDP, we implement all the optimization techniques mentioned in [10]. Since EDP builds its index gradually during the query processing, for fairness, we generate random queries to warm up EDP as [10] until its cache size becomes stable or reaches the memory limit (20GB) before our experiments.

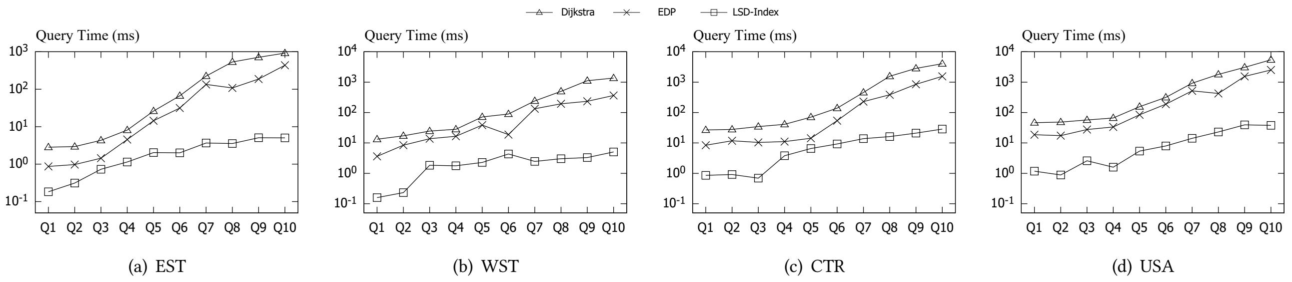
[1]http://users.diag.uniroma1.it/challenge9/download.shtml

Exp-1: Efficiency when varying query distance. In this experiment, we evaluate the query efficiency of the algorithms by varying the label-constrained shortest distance between the source vertex and target vertex in the query. We randomly generate 10 groups of queries $Q_1, \ldots, Q_{10}$ and each group contains 1000 queries. For each query $q = (s, t, \mathcal{L})$ in group i, the label-constrained distance between s and t regarding $\mathcal{L}$ ranging from $(\frac{\delta}{1\text{ km}})^{\frac{i-1}{10}}$ to $(\frac{\delta}{1\text{ km}})^{\frac{i}{10}}$ kilometers, where δ is the longest distance between any two vertices in the road network. And $\mathcal{L}$ is set as minimum edge label set which can make the label-constrained distance between s and t satisfy the above condition. Figure 6 shows the average query processing time for queries in each group on four datasets.

As shown in Figure 6, the query processing time of all the algorithms increases when the distance increases. This is because as the distance between s and t increases, more vertices or nodes have to be explored. Moreover, EDP is always faster than Dijkstra while LSD-Index is much faster than EDP and the performance gap enlarges as the distance increases. The reasons are Dijkstra and EDP have to explore many vertices in the road networks while LSD-Index only needs to visit vertices in the nodes along the tree decomposition, which is much less than that of Dijkstra and EDP.

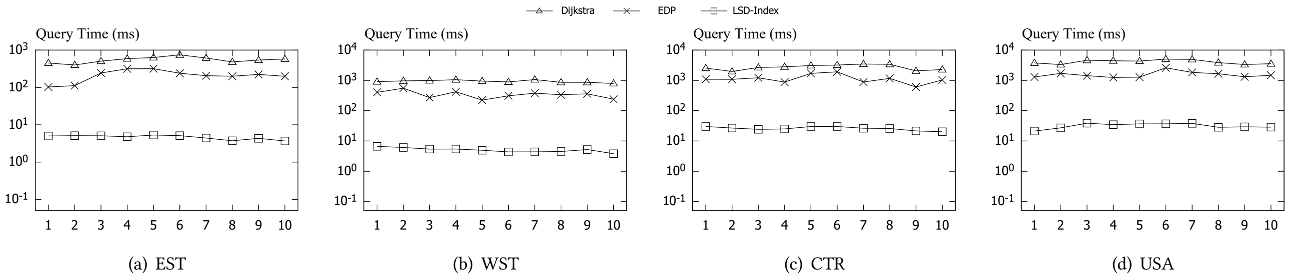
Exp-2: Efficiency when varying $|\mathcal{L}|$. In this experiment, we evaluate the query efficiency of the algorithms by varying $|\mathcal{L}|$ of the queries. To do this, we randomly generate 10 groups of queries and each group contains $1,000$ queries. For each query $q = (s, t, \mathcal{L})$ in group i, $\mathcal{L}$ is set as an edge label set with $|\mathcal{L}| = i$ such that s can reach t following the edges with edge label in $\mathcal{L}$. We record the average query processing time for queries in each group and the results for the four large datasets is demonstrated in Figure 7, the results on the remaining datasets show similar trends.

Based on the results, we can observe that: (1) LSD-Index always outperforms Dijkstra and EDP by at least an order of magnitude. The reasons are the same as discussed in Exp-1. (2) the average processing time of all the algorithms keeps stable when we vary $|\mathcal{L}|$. For Dijkstra and EDP, when $|\mathcal{L}|$ is small, the label-constrained shortest distance between s and t regarding $\mathcal{L}$ is large generally, which implies that the traversal on the road network is long. As $|\mathcal{L}|$ increases, the label-constrained shortest distance between s and t regarding $\mathcal{L}$ becomes small, but the number of edges with edge label in $\mathcal{L}$ increases as well. As a result, the number of explored vertices and edges during the query processing keep similar. For LSD-Index,



(a) EST (b) WST (c) CTR (d) USA

Figure 6: Query Processing Time (Varying Query Distance)



(a) EST (b) WST (c) CTR (d) USA

Figure 7: Query Processing Time (Varying $|\mathcal{L}|$)

LSD-Index processes the queries based on the tree decomposition, and thus the processing is nearly independent with $\mathcal{L}$.

Exp-3: Indexing time. Table 2 presents the time to construct LSD-Index for each dataset, including the sequential construction algorithm and the parallel construction algorithm (running with 32 threads). For the first six road networks, the index can be constructed within 6 minutes even for the sequential construction algorithm. However, the sequential construction algorithm needs 1.5–2 hours to complete the index construction for CTR and USA. Considering the size of these two datasets, the indexing time is acceptable but not highly satisfactory. On the other hand, for the parallel construction algorithm, it takes less than 60 seconds to construct the index for the first six datasets and less than 1, 000 seconds to construct the index for the USA dataset. As shown in the results, our proposed algorithms can efficiently construct LSD-Index in practice, especially the parallel construction algorithm.

Exp-4: Index size. The size of LSD-Index for each road network is shown in Table 2. As shown in Table 2, the index sizes of the first six road networks are within 1 GB, and even for the whole USA road network, the index size is only 2.37 GB. Considering USA dataset is around 0.8 GB in size, 2.37 GB is still small. We omit the index size of EDP because its index size varies by different cache strategies. In our experimental setting, we set the index size limit for EDP to 20 GB and the index sizes for most of the large road networks (WST, CTR, USA) in our setting are beyond 10 GB. From the results, it is clear that LSD-Index is a compact index structure.

Exp-5: Case Study. Figure 8 demonstrates a real-world example of label-constrained shortest path queries. In Sydney, we can briefly divide the roads into three categories: toll road (T), main road (M), and local road (L). Assume that the students from UNSW plan to go to Tarango Zoo by car at weekends. If they only want to get to the zoo as fast as possible, then, they can obtain their route by the query q = (*"UNSW"*, *"Tarango Zoo"*, *"TML"*), which returns p_1 with 15.52km. On the other hand, if they want to get

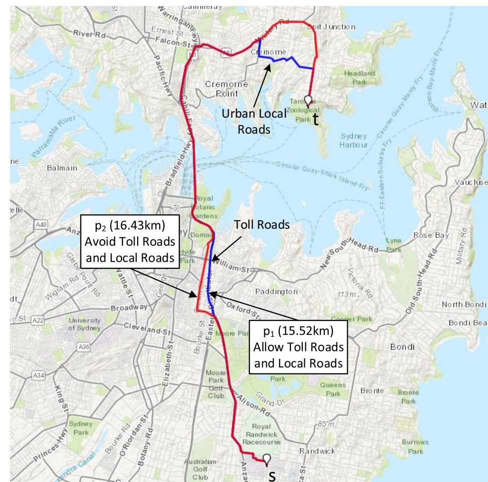


Figure 8: Case Study

to the zoo as fast as possible, but are not willing to pass the toll roads or local roads, they can obtain their route by the query q = (*"UNSW"*, *"Tarango Zoo"*, *"M"*), which return p_2 with 16.43km. From this example, we can see that label-constrained shortest path queries can satisfy different users' requirements in route planning.

Exp-6: Index size when varying $|\Sigma|$. In this experiments, we evaluate the index size when varying $|\Sigma|$. For each datasets, we set the number of labels from $\lceil \frac{|\Sigma|}{5} \rceil$ to $|\Sigma|$. For the smaller label set, we generate them using the similar method mentioned before: we sort the original labels in each main category according to their frequency and merge label labels with similar frequency until the number of labels reach the required size. Figure 9 shows the results.

Figure 9 shows that the index sizes increases as $|\Sigma|$ increases. This is because that the larger $|\Sigma|$ is, the more information needs to be stored in the index. However, even for the largest road network USA, the largest index size is 8.6GB when $|\Sigma| = 32$, which is only

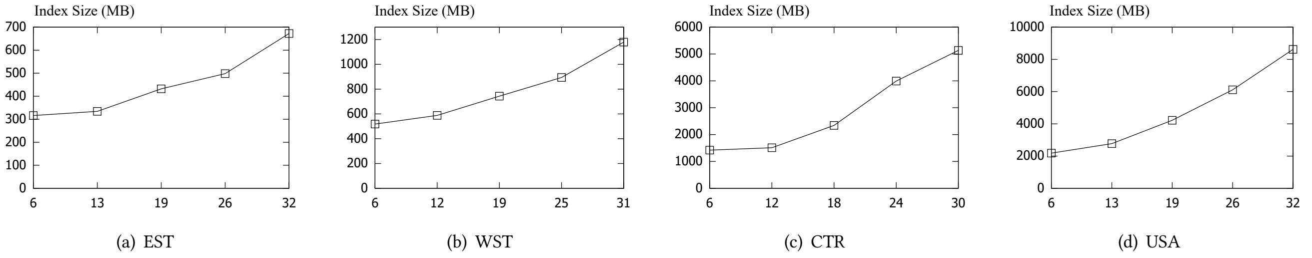


Figure 9: Index Size (Varying $|\Sigma|$)

10 times the size of the dataset (around 0.8GB). The experimental results confirm that LSD-Index is a compact index structure.

8 RELATED WORK

Label-constrained shortest path query. With the proliferation of graph applications, research efforts have been devoted to many fundamental problems in managing and analyzing graph data [6, 18, 19, 23, 33, 36–38]. Among them, label-constrained shortest path query has received considerable attention recently. [25] proposes is a method named CHLR to answer the queries based on CH [8]. EDP [10] is the state-of-the-art algorithm for this problem. The experimental results in [10] show that the performance of EDP is much better than that of CHLR. Therefore, we choose EDP as our baseline. Besides, [4] studies the approximate approaches for label-constrained shorted path queries. As [4] only finds approximate results, the techniques can not be applied to address the problem studied in this paper.

Label-constrained reachability query. Label-constrained reachability query is also studied in the literature. Given a query $q = (s, t, \mathcal{L})$, label-constrained reachability query asks whether s can reach t following edges with label in $\mathcal{L}$. [11] is the first to study this problem. [24, 29, 41] further improve the efficiency of query processing. Since label-constrained reachability queries do not need to consider the specific distance/path information, these techniques are unpromising to be adapted to label-constrained shortest path query problem studied in this paper.

Shortest path/distance query on unlabelled graphs. In the literature, a plethora of indexing techniques have been proposed to answer the shortest path/distance queries on unlabelled graphs, such as [1, 8, 12, 21, 30, 40]. [32] and [17] evaluate the practical performance of the representative algorithms. However, as discussed in Section 4, since these methods do not consider the edge labels, direct extending these methods to address the label-constrained shortest path will lead to exponential number of index structures, which makes this approach inapplicable in practice.

It is noteworthy that the success of H2H [21] inspires us to revisit the tree decomposition for the label-constrained shortest path query problem. H2H is designed for shortest distance queries on unlabeled road networks and it also builds its index based on the tree decomposition. For each tree node (or corresponding vertex), H2H pre-computes and stores a distance array for the vertices in its ancestors. H2H answers a distance query by iterating distances from the source/destination vertices to the vertices in LCA, and the query processing time can be bounded by $O(\omega)$.

Although the high level idea of H2H seems similar to ours, H2H is significantly different from our proposed algorithm in Section 4 (Algorithm 1) in the following four aspects: (1) H2H cannot be easily adapted to answer the shortest path queries on unlabeled road networks, not to mention the label-constrained path queries studied in this paper. Since H2H is tailored for fast shortest distance query, the path information is reduced as much as possible in H2H. Consequently, it is very hard for H2H to restore the corresponding path based on the computed shortest distance. However, our index natively supports shortest path queries. (2) The query processing algorithm of H2H is also different from ours. H2H answers a distance query by first locating the LCA of the nodes representing the source and destination vertices and then joining the distances from source/destination vertices to vertices in LCA, while our approach have to iterate up along the tree from nodes representing the source and destination vertices to their LCA. (3) We cannot directly prove that the query processing time of Algorithm 1 can be bounded $O(h \cdot \omega)$ (Theorem 4.13) based on the theoretical results of H2H. The proof of Theorem 4.13 is based on Lemma 4.11 and Lemma 4.12. For Lemma 4.11, we can easily obtain a similar conclusion from H2H with Property 2 and Definition 5.1 in [21]. However, for Lemma 4.12, no similar conclusion can be easily derived according to the theoretical results of [21]. Since Lemma 4.12 is the most essential lemma for the proof of Theorem 4.13, our theoretical findings cannot be easily obtained based on [21]. (4) The index size of H2H is much larger than ours. In each tree node, H2H stores distances to the vertices representing by all its ancestor nodes, the size of the index could be very large. As shown in the experiment of [21], the H2H index size for USA road network is over 100GB, which is much larger than ours.

9 CONCLUSION

In this paper, we study the label-constrained shortest path query problem on road networks. We devise a novel index structure named LSD-Index based on the tree decomposition. With LSD-Index, we propose an efficient query processing algorithm to answer the queries. Moreover, we also present efficient index construction algorithms. The experimental results demonstrate the efficiency of our proposed algorithms. For future work, we are interested in extending our work to dynamic graphs by devising efficient index maintenance algorithm for graph label/vertex/edge updates.

ACKNOWLEDGMENTS

Long Yuan is supported by NSFC61902184, NSF of Jiangsu Province BK20190453, and Science and Technology on Information Systems Engineering Laboratory WDZC20205250411. Lu Qin is supported by ARC FT200100787 and ARC DP210101347. Ying Zhang is supported by ARC FT170100128 and ARC DP210101393.

REFERENCES

- [1] Takuya Akiba, Yoichi Iwata, Ken-ichi Kawarabayashi, and Yuki Kawata. 2014. Fast shortest-path distance queries on road networks by pruned highway labeling. In *Proceedings ALENEX*. SIAM, 147–154.
- [2] Stefan Arnborg, Derek G Corneil, and Andrzej Proskurowski. 1987. Complexity of finding embeddings in ak-tree. *SIAM Journal on Algebraic Discrete Methods* 8, 2 (1987), 277–284.
- [3] Hannah Bast, Daniel Delling, Andrew V. Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. 2016. Route Planning in Transportation Networks. In *Algorithm Engineering - Selected Results and Surveys*. Lecture Notes in Computer Science, Vol. 9220. 19–80. https://doi.org/10.1007/978-3-319-49487-6_2
- [4] Francesco Bonchi, Aristides Gionis, Francesco Gullo, and Antti Ukkonen. 2014. Distance oracles in edge-labeled graphs. In *Proceedings of EDBT*. OpenProceedings.org, 547–558. https://doi.org/10.5441/002/edbt.2014.49
- [5] Lijun Chang, Jeffrey Xu Yu, Lu Qin, Hong Cheng, and Miao Qiao. 2012. The exact distance to destination in undirected world. *VLDB J.* 21, 6 (2012), 869–888. https://doi.org/10.1007/s00778-012-0274-x
- [6] Zi Chen, Long Yuan, Xuemin Lin, Lu Qin, and Jianye Yang. 2020. Efficient Maximal Balanced Clique Enumeration in Signed Networks. In *Proceedings of The Web Conference 2020*. 339–349.
- [7] EW Djikstra. 1959. A note on two problems in connection with graphs. *Numer. Math.* 1 (1959), 269–271.
- [8] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. 2008. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In *Proceedings of WEA*. 319–333.
- [9] Andrew V Goldberg and Chris Harrelson. 2005. Computing the shortest path: A search meets graph theory. In *Proceedings of SODA*. 156–165.
- [10] Mohamed S. Hassan, Walid G. Aref, and Ahmed M. Aly. 2016. Graph Indexing for Shortest-Path Finding over Dynamic Sub-Graphs. In *Proceedings of SIGMOD*. ACM, 1183–1197. https://doi.org/10.1145/2882903.2882933
- [11] Ruoming Jin, Hui Hong, Haixun Wang, Ning Ruan, and Yang Xiang. 2010. Computing label-constraint reachability in graph databases. In *Proceedings of SIGMOD*. ACM, 123–134. https://doi.org/10.1145/1807167.1807183
- [12] Sungwon Jung and Sakti Pramanik. 2002. An Efficient Path Computation Model for Hierarchically Structured Topographical Road Maps. *IEEE TKDE* 14, 5 (2002), 1029–1046.
- [13] Arie M. C. A. Koster, Hans L. Bodlaender, and Stan P. M. van Hoesel. 2001. Treewidth: Computational Experiments. *Electron. Notes Discret. Math.* 8 (2001), 54–57. https://doi.org/10.1016/S1571-0653(05)80078-2
- [14] Huayu Li, Yong Ge, Richang Hong, and Hengshu Zhu. 2016. Point-of-Interest Recommendations: Learning Potential Check-ins from Friends. In *Proceedings of SIGKDD*. ACM, 975–984. https://doi.org/10.1145/2939672.2939767
- [15] Wentao Li, Miao Qiao, Lu Qin, Ying Zhang, Lijun Chang, and Xuemin Lin. 2020. Scaling Up Distance Labeling on Graphs with Core-Periphery Properties. In *Proceedings of SIGMOD*. ACM, 1367–1381. https://doi.org/10.1145/3318464.3389748
- [16] YiRu Li, Sarah George, Craig Apfelbeck, Abdeltawab M. Hendawi, David Hazel, Ankur Teredesai, and Mohamed H. Ali. 2014. Routing service with real world severe weather. In *Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, Dallas/Fort Worth, TX, USA, November 4-7, 2014*, Yan Huang, Markus Schneider, Michael Gertz, John Krumm, and Jagan Sankaranarayanan (Eds.). ACM, 585–588. https://doi.org/10.1145/2666310.2666375
- [17] Ye Li, Leong Hou U, Man Lung Yiu, and Ngai Meng Kou. 2017. An Experimental Study on Hub Labeling based Shortest Path Algorithms. *Proc. VLDB Endow.* 11, 4 (2017), 445–457. https://doi.org/10.1145/3186728.3164141
- [18] Boge Liu, Long Yuan, Xuemin Lin, Lu Qin, Wenjie Zhang, and Jingren Zhou. 2019. Efficient (α, β)-core computation: An index-based approach. In *The World Wide Web Conference*. 1130–1141.
- [19] Boge Liu, Long Yuan, Xuemin Lin, Lu Qin, Wenjie Zhang, and Jingren Zhou. 2020. Efficient (α, β)-core computation in bipartite graphs. *The VLDB Journal* 29, 5 (2020), 1075–1099.
- [20] Yiding Liu, Tuan-Anh Pham, Gao Cong, and Quan Yuan. 2017. An Experimental Evaluation of Point-of-interest Recommendation in Location-based Social Networks. *Proc. VLDB Endow.* 10, 10 (2017), 1010–1021. https://doi.org/10.14778/3115404.3115407
- [21] Dian Ouyang, Lu Qin, Lijun Chang, Xuemin Lin, Ying Zhang, and Qing Zhu. 2018. When Hierarchy Meets 2-Hop-Labeling: Efficient Shortest Distance Queries on Road Networks. In *Proceedings of SIGMOD*. ACM, 709–724. https://doi.org/10.1145/3183713.3196913
- [22] Dian Ouyang, Long Yuan, Lu Qin, Lijun Chang, Ying Zhang, and Xuemin Lin. 2020. Efficient Shortest Path Index Maintenance on Dynamic Road Networks with Theoretical Guarantees. *Proc. VLDB Endow.* 13, 5 (2020), 602–615. https://doi.org/10.14778/3377369.3377371
- [23] Dian Ouyang, Long Yuan, Fan Zhang, Lu Qin, and Xuemin Lin. 2018. Towards efficient path skyline computation in bicriteria networks. In *International Conference on Database Systems for Advanced Applications*. Springer, 239–254.
- [24] You Peng, Ying Zhang, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2020. Answering Billion-Scale Label-Constrained Reachability Queries within Microsecond. *Proc. VLDB Endow.* 13, 6 (2020), 812–825. https://doi.org/10.14778/3380750.3380753
- [25] Michael N. Rice and Vassilis J. Tsotras. 2010. Graph Indexing of Road Networks for Shortest Path Queries with Label Restrictions. *Proc. VLDB Endow.* 4, 2 (2010), 69–80. https://doi.org/10.14778/1921071.1921074
- [26] Neil Robertson and Paul D. Seymour. 1984. Graph minors. III. Planar tree-width. *J. Comb. Theory, Ser. B* 36, 1 (1984), 49–64. https://doi.org/10.1016/0095-8956(84)90013-3
- [27] Neil Robertson and Paul D. Seymour. 1986. Graph Minors. II. Algorithmic Aspects of Tree-Width. *J. Algorithms* 7, 3 (1986), 309–322. https://doi.org/10.1016/0196-6774(86)90023-4
- [28] Peter Sanders and Dominik Schultes. 2005. Highway Hierarchies Hasten Exact Shortest Path Queries. In *Proceedings of ESA*. 568–579.
- [29] Lucien D. J. Valstar, George H. L. Fletcher, and Yuichi Yoshida. 2017. Landmark Indexing for Evaluation of Label-Constrained Reachability Queries. In *Proceedings of SIGMOD*. ACM, 345–358. https://doi.org/10.1145/3035918.3035955
- [30] Fang Wei. 2010. TEDI: efficient shortest path query answering on graphs. In *Proceedings of SIGMOD*. ACM, 99–110. https://doi.org/10.1145/1807167.1807181
- [31] Wikipedia. 2021. Expressways of China. https://en.wikipedia.org/wiki/Expressways_of_China
- [32] Lingkun Wu, Xiaokui Xiao, Dingxiong Deng, Gao Cong, Andy Diwen Zhu, and Shuigeng Zhou. 2012. Shortest Path and Distance Queries on Road Networks: An Experimental Evaluation. *Proc. VLDB Endow.* 5, 5 (2012), 406–417. https://doi.org/10.14778/2140436.2140438
- [33] Xudong Wu, Long Yuan, Xuemin Lin, Shiyu Yang, and Wenjie Zhang. 2019. Towards efficient k-tripeak decomposition on large graphs. In *International Conference on Database Systems for Advanced Applications*. Springer, 604–621.
- [34] Guochang Xu and Yan Xu. 2016. *GPS: theory, algorithms and applications*. Springer.
- [35] Jinbo Xu, Feng Jiao, and Bonnie Berger. 2005. A tree-decomposition approach to protein structure prediction. In *Proceedings of CSB*. IEEE, 247–256.
- [36] Long Yuan, Lu Qin, Xuemin Lin, Lijun Chang, and Wenjie Zhang. 2016. Diversified top-k clique search. *The VLDB Journal* 25, 2 (2016), 171–196.
- [37] Long Yuan, Lu Qin, Xuemin Lin, Lijun Chang, and Wenjie Zhang. 2017. Effective and efficient dynamic graph coloring. *Proceedings of the VLDB Endowment* 11, 3 (2017), 338–351.
- [38] Long Yuan, Lu Qin, Wenjie Zhang, Lijun Chang, and Jianye Yang. 2017. Index-based densest clique percolation community search in networks. *IEEE Transactions on Knowledge and Data Engineering* 30, 5 (2017), 922–935.
- [39] Junhua Zhang, Long Yuan, Wentao Li, Lu Qin, and Ying Zhang. 2021. Technical Report. https://www.dropbox.com/sh/91ctwdxinybjyr9/AABfJJ8oUk-vjCwYz-KIsJ0_a?dl=0
- [40] Andy Diwen Zhu, Hui Ma, Xiaokui Xiao, Siqiang Luo, Youze Tang, and Shuigeng Zhou. 2013. Shortest path and distance queries on road networks: towards bridging theory and practice. In *Proceedings of SIGMOD*. 857–868.
- [41] Lei Zou, Kun Xu, Jeffrey Xu Yu, Lei Chen, Yanghua Xiao, and Dongyan Zhao. 2014. Efficient processing of label-constraint reachability queries in large graphs. *Inf. Syst.* 40 (2014), 47–66. https://doi.org/10.1016/j.is.2013.10.003